View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositori Institucional de la Universitat Jaume I

Tercer boletín de prácticas sobre MMPor

IG29: Compiladores e intérpretes

Quinta sesión de prácticas

(1) Experimenta con el siguiente programa para ver cuál es la información que automáticamente se almacena en la lista `sys.argv` al llamarlo de diferentes formas:

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-

import sys

n=len(sys.argv)
for i in range(n):
    print "sys.argv[%d]: %s" % (i,sys.argv[i])

print "Número de argumentos: %d" % (n-1)
```

Concretamente:

- ¿El resultado de llamar directamente al programa es diferente del de pasarlo como primer argumento de la orden `python`?
- ¿Se guarda el nombre del programa en `sys.argv`?
- ¿Importa con qué ruta se le llame?
- ¿Se separan correctamente los diferentes argumentos de la línea de órdenes?
- ¿Las redirecciones se consideran argumentos o no es así?

(2) Ahora que ya debes saber manejar `sys.argv`, adapta la implementación de tu calculadora para que acepte una opción `-s` destinada a mostrar el AST de la entrada (en vez de ejecutarlo). Concretamente, has de modificar la función `main` con los siguientes objetivos:

- Si se llama al programa con más de un argumento o éste no es `-s`, debe utilizarse `trata_error` para proporcionar un mensaje adecuado, indicando que el *`lugar`* es la línea de órdenes.
- Si la llamada es correcta, debe guardarse en una variable si el programa ha de funcionar en modo normal o `-s`.
- Posteriormente, en función del valor de esa variable, debe decidirse si el AST resultante del análisis de la entrada hay que ejecutarlo o imprimirlo.

Prueba el programa resultante llamándolo con dos argumentos (error), con una opción `-x` (error), sin argumentos (modo normal) y con la opción `-s`.

(3) Modifica los métodos `__str__` de los diferentes nodos de tu representación semántica para conseguir que el formato de impresión de los ASTs coincida con el que verArbol acepta como entrada. Por ejemplo, el AST para la entrada

```
a:= 2+3*4
b:= a-1-2
```

debería mostrarse como algo parecido a esto:

```
( "Secuencia"
    ( "Asigna" "lid: a"
        ( "Mas" ( "Cte" "v: 2" )
                ( "Por" ( "Cte" "v: 3" ) ( "Cte" "v: 4" ) )
        )
    )
    ( "Asigna" "lid: b"
```

```
            ( "Menos" ( "Menos" ( "Vble" "lid: a" ) ( "Cte" "v: 1" ) )
                      ( "Cte" "v: 2" )
            )
        )
    )
```

a. Puedes conseguir cómodamente los métodos para `Mas`, `Menos` y `Por` heredándolos de una nueva clase auxiliar, `Binaria`, definida como sigue:

```
class Binaria: # clase auxiliar para herencias

    def __str__(self):
        nombre=self.__class__.__name__
        return '( "%s" %s %s )' % (nombre,self.hi,self.hd)
```

Elimina los antiguos métodos `__str__` y define las clases como `Mas(Binaria)`, `Menos(Binaria)` y `Por(Binaria)`. Considera la posibilidad de que los correspondientes métodos constructores también se hereden de `Binaria`.

b. Para los nodos Cte, Vble y Asigna, no olvides representar los correspondientes pares atributo/valor mediante cadenas auxiliares.

Y, si seguiste correctamente las prácticas anteriores, ¡el método `__str__` de Secuencia ya estará preparado para seguir el formato verArbol!

(4) Prueba diferentes aspectos del estado actual de tu programa:

a. Utiliza tus ficheros de prueba para comprobar que, al ser llamado sin argumentos, `mmpor` sigue efectuando los mismo cálculos que antes.

b. Prueba con opción `-s` y redirección a un fichero `ast.vA` la entrada sugerida anteriormente:

```
a:= 2+3*4
b:= a-1-2
```

¿Parece correcta la salida obtenida en el fichero?

c. Ejecuta verArbol pasándole `ast.vA` por la entrada estándar. ¿Se ve correctamente el AST? Coteja lo que ves con lo que, según tu diseño, deberías ver:

- ¿Los nodos se muestran con los mismos nombres que documentaste en el diseño[1]?
- ¿Sucede lo mismo con los atributos? ¿Hay alguno que no se muestre cuando sitúas el cursor sobre el nodo correspondiente?

d. Revisa los ASTs correspondientes a tus ficheros de prueba. Utiliza tuberías para ello, como se muestra a continuación para un supuesto fichero `prueba.mmp`[2]:

```
./mmpor -s < prueba.mmp | ./verArbol
```

(5) Entrega en un paquete `entrega04.tgz` los ficheros siguientes:

- `disenyo.txt`.
- `mmpor.mc`, `AST.py`, `memo.py` y `errores.py`.
- Los correspondientes ficheros de prueba.

(6) Puedes ir adelantando trabajo si piensas en cómo modificar tu diseño y tu implementación para conseguir lo siguiente:

- Cada línea podrá ser una asignación (que ya no supondrá impresión alguna) o bien una sentencia de escritura por la salida estándar.
- Las sentencias de escritura consistirán en una palabra reservada, `print`, seguida de una secuencia de uno o más elementos separados por comas donde cada elemento podrá ser una expresión o bien un literal de cadena (secuencia delimitada por caracteres doble comilla y sin tabuladores, saltos de línea ni dobles comillas en su interior).
- Los valores de los elementos de una sentencia de escritura se imprimirán sin separación entre ellos y la ejecución de la sentencia finalizará con la impresión de un salto de línea.

---

1. No consideraremos relevante que no se muestren acentos.

2. En realidad, habrás de especificar los nombres de los programas con sus correspondientes rutas, o bien crear enlaces simbólicos adecuados en tu directorio de pruebas.